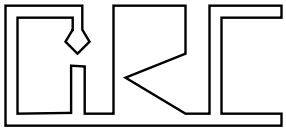

Dorel Lucanu
Faculty of Computer Science
Alexandru Ioan Cuza University, Iaşi, Romania, dlucanu@info.uaic.ro

Grigore Roşu
Department of Computer Science
University of Illinois at Urbana-Champaign, USA, grosu@cs.uiuc.edu

CIRC User Manual

CIRC Prover User Manual

Dorel Lucanu, Grigore Roşu

Version 1.0

1 Getting Started

1.1 Installing

To run CIRC tool you need to have installed Maude version 2.2 or later. Maude tool can be downloaded from Maude home page `http://maude.cs.uiuc.edu/`. The CIRC tool can be downloaded from CIRC home page `http://fsl.cs.uiuc.edu/index.php/Circ` or `http://circ.info.uaic.ro/`.

1.2 Loading

To load the prover, introduce the including command `in circ`:

```
Maude> in circ
==========================================
fmod BASIC-DATA-TYPES
==========================================
fmod DATA-TYPES
==========================================
fmod EQ-MANAGER
==========================================
fmod PRE-OUTPUT
==========================================
mod CiRC-DATABASE-HANDLING
==========================================
mod PROVER
==========================================
fmod CiRC-COMMAND-SIGN
==========================================
fmod META-CiRC-SIGN
==========================================
fmod BASIC-PARSER
==========================================
mod PARSER
==========================================
mod CIRC

Maude>
```

1.3 Commands

The general syntax for a command is

$$(\langle command \rangle)$$

where

$$\begin{aligned}\langle command\rangle ::= & \langle specification\rangle \\ & \langle addGoal\rangle \\ & \langle setFirstGoal\rangle \\ & \langle showGoals\rangle \\ & \langle addLemma\rangle \\ & \langle showLemmas\rangle \\ & \langle removeLemmas\rangle \\ & \langle eqReduction\rangle \\ & \langle coinduction\rangle \\ & \langle induction\rangle \\ & \langle caseAnalysis\rangle \\ & \langle removeHypothesis\rangle \\ & \langle showHistory\rangle \\ & \langle clearHistory\rangle \\ & \langle quitProof\rangle\end{aligned}$$

Before introducing the first command, you must select CIRC as the active module and initialize the state of the loop mode:

```
Maude> select CIRC .
Maude> loop init .
```

The two commands are included in the file `startCIRC.maude` so you can use the following shorter way to activate CIRC prover:

```
Maude> in startCIRC .
```

In what follows we use by default the following definition:

$$\langle itemList\rangle ::= \langle item\rangle \mid \langle item\rangle\ \langle itemList\rangle$$

where *item* is instantiated over different syntactical constructions.

1.3.1 Introducing a specification

This is the first command which must be introduced when you want to use CIRC for proving coinductive/inductive properties. This command has a double function: 1. it includes information regarding the (behavioural) specification, derivatives, case analysis, and 2. it starts the prover by creating the initial configuration.

The syntax of this command is:

$$\begin{aligned}\langle specification\rangle ::= & \texttt{cmod}\ \langle declarationList\rangle\ \texttt{endcm} \\ \langle declaration\rangle ::= & \langle importing\rangle \mid \\ & \langle derivative\rangle \mid \\ & \langle cases\rangle\end{aligned}$$

where importing declaration $\langle importing\rangle$ is as in Maude, a derivative declaration has the syntax

$$\langle derivative\rangle ::= \texttt{derivative}\ \langle termList\rangle$$

and declares a set of derivatives where the placeholder of the state is pointed by a star-variable $*{:}\langle sort\rangle$, and a case specification has the syntax

$$\langle cases\rangle ::= \texttt{cases}\ \langle caseName\rangle\ \texttt{are}\ \langle caseExprList\rangle$$
$$\langle caseExpr\rangle ::= \langle term\rangle\ \texttt{=}\ \langle term\rangle$$

A $\langle specification\rangle$ command must include just one importing statement, zero or more derivative declarations, zero or more cases declarations.

Example: Streams.

```
( cmod B-STREAM is
    importing STREAM .
    derivative hd(*:Stream) .
    derivative tl(*:Stream) .
  endcm
)
```

The above command is equivalent to

```
( cmod B-STREAM is
    importing STREAM .
    derivative hd(*:Stream) tl(*:Stream) .
  endcm
)
```

1.3.2 Adding a goal

Once the specification was introduced, it can be checked if it has the desired properties. A property to be checked is called *goal*. The command for introducing goals has the following syntax:

$$\begin{aligned}
\langle addGoal\rangle &::= \texttt{add goal}\ \langle equation\rangle\ \texttt{.} \mid \\
&\qquad\quad \texttt{add cgoal}\ \langle condEquation\rangle\ \texttt{.} \\
\langle equation\rangle &::= \langle term\rangle\ \texttt{=}\ \langle term\rangle \\
\langle condEquation\rangle &::= \langle term\rangle\ \texttt{=}\ \langle term\rangle\ \texttt{if}\ \langle condition\rangle \\
\langle condition\rangle &::= \langle term\rangle\ \texttt{=}\ \langle term\rangle \mid \langle condition\rangle\ /\backslash\ \langle condition\rangle
\end{aligned}$$

Example: Streams.

```
( add goal zip(odd(S:Stream), even(S:Stream)) = S:Stream .)
```

1.3.3 Showing goals

Displays the goals to be proved. Its syntax is

$$\langle showGoals\rangle ::= \texttt{show goals .}$$

Example: Streams.

```
Maude> (show goals .)
zip(odd(S:Stream),even(S:Stream))= S:Stream
Maude>
```

1.3.4 Set first goal

Changes the order of the goals by moving a given goal on the first position. To see the order of the goals use `show goals` command. The syntax is

$$\langle setFirstGoal \rangle ::= \texttt{set first goal } \langle number \rangle \texttt{ .}$$

Example: Streams.

```
Maude> ( add goal odd(zip(S:Stream, S':Stream)) = S:Stream .)
Goal odd(zip(S:Stream,S':Stream))= S:Stream   added.

Maude> ( add goal map-f(iter-f(E:Elt)) = iter-f(f(E:Elt)) .)
Goal map-f(iter-f(E:Elt))= iter-f(f(E:Elt))  added.

Maude> (show goals .)
1 .   map-f(iter-f(E:Elt))= iter-f(f(E:Elt))
2 .   odd(zip(S:Stream,S':Stream))= S:Stream
3 .   zip(odd(S:Stream),even(S:Stream))= S:Stream

Maude> (set first goal 3 .)

Maude> (show goals .)
1 .   zip(odd(S:Stream),even(S:Stream))= S:Stream
2 .   map-f(iter-f(E:Elt))= iter-f(f(E:Elt))
3 .   odd(zip(S:Stream,S':Stream))= S:Stream
Maude>
```

1.3.5 Adding a lemma

Sometimes a lemma is needed to prove a goal. This lemma could be proved priory, on-the-fly, or subsequently. The command for introducing lemmas has the following syntax:

$$\begin{aligned}\langle addLemma \rangle ::= &\texttt{ add lemma } \langle equation \rangle \texttt{ . } | \\ &\texttt{add clemma } \langle condEquation \rangle \texttt{ .}\end{aligned}$$

Example: Finite Lists.

```
Maude> (add lemma E:Elt L:MyList = cons(E:Elt, L:MyList) .)
Lemma E:Elt L:MyList  =  cons(E:Elt,L:MyList) added.
```

1.3.6 Showing lemmas

Displays all the lemmas added during the proving. Its syntax is:

$$\langle showLemmas \rangle ::= \texttt{show lemmas .}$$

Example: Finite Lists.

```
Maude> (show lemmas .)
E:Elt L:MyList  =  cons(E:Elt,L:MyList)
```

1.3.7 Removing lemmas

Removes all the lemmas added during the proving. Its syntax is:

$$\langle removeLemmas \rangle ::= \texttt{remove lemmas .}$$

Example: Finite Lists.

```
Maude> (remove lemmas .)
All lemmas removed.

Maude> (show lemmas .)

Maude>
```

1.3.8 Equation reduction

Tries to prove the first goal by equational reduction. This command is useful in combination with `induction` and `coinduction` commands. Its syntax is:

$$\langle eqReduction \rangle ::= \texttt{reduce .}$$

1.3.9 Coinduction.

Starts the algorithm of circular coinduction over the set of goals using the derivatives included in the specification. Its syntax is

$$\langle coinduction \rangle ::= \texttt{coinduction of depth } \langle number \rangle \texttt{ .}$$

If the parameter of the command is a positive number n, then the algorithm stops after n executions of the Circular Coinduction rule. The bound n is useful to avoid an eventual infinite run of the algorithm. If $n = -1$, then the execution of the algorithm is unbound. Usually, the algorithm successfully ends or fails after a finite number of steps. But there are cases when the number of circularities generated are infinite.

Example: Streams.

```
Maude> (show goals .)
zip(odd(S:Stream),even(S:Stream))= S:Stream
Maude> (coinduction of depth 1 .)
Hypothesis zip(odd(S:Stream),even(S:Stream))= S:Stream   added.

Maude> (show goals .)
1 .   hd zip(odd(S:Stream),even(S:Stream))= hd S:Stream
2 .   tl(zip(odd(S:Stream),even(S:Stream)))= tl(S:Stream)

Maude> (reduce .)
Goal hd zip(odd(S:Stream),even(S:Stream))= hd S:Stream   proved by reduction.

Maude> (show goals .)
tl(zip(odd(S:Stream),even(S:Stream)))= tl(S:Stream)
Maude> (coinduction of depth -1 .)
Proof succeeded.

Maude>
```

1.3.10 Induction.

Starts the algorithm of circular induction over the set of goals. Its syntax is

$$\langle induction \rangle ::= \texttt{induction on } \langle variable \rangle \texttt{ of depth } \langle number \rangle \texttt{ .}$$

The variable must be specified as a pair `name:Sort`. The number parameter has a similar meaning as that for coinduction.

Example: Finite trees.

```
Maude> in startCIRC .
Maude> (cmod BTREE is
          importing TREE .
        endcm )
Prover started.
Maude> (add goal length(L1:TList ; L2:TList) =
                 length(L1:TList) + length(L2:TList) .)
Goal length(L1:TList ; L2:TList)= length(L1:TList)+ length(L2:TList)  added.
Maude> (induction on L1:TList of depth -1 .)
Proof succeeded.
Maude>
```

1.3.11 Case analysis

This command is useful when the specification includes conditional equations or some operations are underspecified. The proving algorithms are interactively instructed to process separately each case. The syntax is

⟨*caseAnalysis*⟩ ::= `case analysis on` ⟨*caseName*⟩ `with` ⟨*substitution*⟩

⟨*substitution*⟩ ::= ⟨*assignmentList*⟩

⟨*assignment*⟩ ::= ⟨*variable*⟩ `<-` ⟨*term*⟩

The parameters of the command specify the name of the case definition and the substitution used for analysis.

Example: Infinite lists and finite lists.

```
Maude> ( add goal app(L:List, app(L':List, L'':List)) =
                  app(app(L:List,L':List),L'':List) .)
Goal app(L:List,app(L':List,L'':List))= app(app(L:List,L':List),L'':List)
    added.

Maude> (coinduction of depth -1 .)
Visible goal hd(app(L:List,app(L':List,L'':List)))= hd(app(app(L:List,L':List),
    L'':List))  failed during coinduction.

Maude> (case analysis on BOOL with N?:Nat? <- hd(L:List) .)
Goal hd(app(L:List,app(L':List,L'':List)))= hd(app(app(L:List,L':List),
     L'':List))  splitted into cases according to case sentence BOOL with
     substitution N?:Nat? <- hd(L:List) .

Maude> (show goals .)
1 .   hd(app(L:List,app(L':List,L'':List)))= hd(app(app(L:List,L':List),
    L'':List))  if error?(hd(L:List))= false
2 .   hd(app(L:List,app(L':List,L'':List)))= hd(app(app(L:List,L':List),
    L'':List))  if hd(L:List)= err
3 .   tl(app(L:List,app(L':List,L'':List)))= tl(app(app(L:List,L':List),
    L'':List))

Maude> (reduce .)
Goal hd(app(L:List,app(L':List,L'':List)))= hd(app(app(L:List,L':List),
     L'':List))  if error?(hd(L:List))= false   proved by reduction.
```

```
Maude> (reduce .)
Goal hd(app(L:List,app(L':List,L'':List)))= hd(app(app(L:List,L':List),
     L'':List))  if hd(L:List)= err   reduced to
     hd(app(L':List,L'':List))= hd(app(app(L:List,L':List),L'':List))
     if hd(L:List)= err
...
```

1.3.12 Remove hypotheses

This command removes the hypotheses added during the current proof(s). It is useful when you wish to prove a new goal with the initial specification, without the hypotheses added by the previous proofs. Its syntax is

$$\langle removeHypotheses \rangle ::= \texttt{remove hypotheses .}$$

Note that the lemmas previously added are not removed by this command. The lemmas are removed with `remove lemmas` command.

Example: Streams.

```
Maude> ( add goal zip(odd(S:Stream), even(S:Stream)) = S:Stream .)
Goal zip(odd(S:Stream),even(S:Stream))= S:Stream   added.

Maude> (coinduction of depth -1 .)
Proof succeeded.

Maude> (remove hypotheses .)

Maude> ( add goal odd(zip(S:Stream, S':Stream)) = S:Stream .)
Goal odd(zip(S:Stream,S':Stream))= S:Stream   added.

Maude> (coinduction of depth -1 .)
Proof succeeded.

Maude>
```

1.3.13 Show history

This command displays the main steps performed by the algorithm in the current proofs. Its syntax is

$$\langle showHistory \rangle ::= \texttt{show history .}$$

Example: Streams.

```
Maude> (show history .)
Goal zip(odd(S:Stream),even(S:Stream))= S:Stream   added.
Hypothesis zip(odd(S:Stream),even(S:Stream))= S:Stream   added.
Goal hd zip(odd(S:Stream),even(S:Stream))= hd S:Stream   proved by reduction.
Hypothesis zip(even(S:Stream),even(tl(S:Stream)))= tl(S:Stream)  added.
Goal hd zip(even(S:Stream),even(tl(S:Stream)))= hd tl(S:Stream)  proved by
    reduction.
Hypothesis zip(even(tl(S:Stream)),odd(tl(tl(tl(S:Stream)))))= tl(tl(S:Stream))
    added.
Goal hd zip(even(tl(S:Stream)),odd(tl(tl(tl(S:Stream)))))= hd tl(tl(S:Stream))
    proved by reduction.
```

```
Hypothesis zip(odd(tl(tl(tl(S:Stream)))),odd(tl(tl(tl(tl(S:Stream))))))=
    tl(tl(tl(S:Stream)))  added.
Goal hd zip(odd(tl(tl(tl(S:Stream)))),odd(tl(tl(tl(tl(S:Stream))))))=
    hd tl(tl(tl(S:Stream)))  proved by reduction.
Goal tl(zip(odd(tl(tl(tl(S:Stream)))),odd(tl(tl(tl(tl(S:Stream)))))))=
    tl(tl(tl(tl(S:Stream))))  proved by reduction.
Proof succeeded.

Maude>
```

1.3.14 Quit proof

This command quits the current proof(s) and removes the information regarding the proof(s) including the hypotheses and lemmas added during the current proof(s). It is useful when the current proof fails or you do not know how to continue it and you wish to start a new proof with the initial specification. Its syntax is

$$\langle quitProof \rangle ::= \texttt{quit proof .}$$

Example: Streams.

```
Maude> ( add goal zip(S:Stream, zip(S':Stream, S'':Stream)) =
                  zip(zip(S:Stream, S':Stream), S'':Stream) .)
Goal zip(S:Stream,zip(S':Stream,S'':Stream))= zip(zip(S:Stream,S':Stream),
    S'':Stream)  added.

Maude> (coinduction of depth -1 .)
Visible goal hd S':Stream = hd S'':Stream   failed during coinduction.

Maude> (quit proof .)
Proving process cacelled.
All hypotheses and lemmas removed.

Maude> (show goals .)

Maude> (show history .)

Maude> (show lemmas .)

Maude>
```

2 Cautions

1. Avoid to use operation names starting with `fr&`.
2. Avoid to use constant names starting with `CC-V2C`.
3. Avoid to use variable names of the form `...&...#`$\langle number \rangle$....
4. It is highly recommended to not declare constructors if the induction is not used in the proving process. Since the constructor variable are frozen, they cannot be properly instantiated when the coinductive hypothesis is applied. Note that the built-in modules NAT and INT include constructor declarations.